修士学位論文

題目

Detection of Fault Introduced by Change Inconsistency In Code Clones

(コードクローンに対する一貫性のない変更に起因する欠陥の検出)

指導教員
井上 克郎 教授

報告者
Yii Yong Lee

平成 20 年 2 月 8 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 19 年度 修士学位論文

Detection of Fault Introduced by Change Inconsistency In Code Clones
コードクローンに対する一貫性のない変更に起因する欠陥の検出

Yii Yong Lee

Abstract

Large software tends to have a significant amount of similar code, commonly known as code clones. Often the code clones are introduced through copy-and-paste process for code reuse purpose where the pasted code usually will go through some modifications such as renaming of identifiers and changing of parameters. When these modifications are done manually, there is possibility that the renaming or changing is not completely applied to all relevant instances by mistake, therefore introducing unintended inconsistency which is considered as bug to the system.

In this thesis, we propose a method using token comparison approach to detect bugs caused by abovementioned modifications. Our method tokenizes detected clones and performs a comparison to find out inconsistencies between them. The inconsistencies are then ranked based on predefined metrics to produce a bug candidate list. We implemented our method and evaluated on open source project. Our tool has found real bugs from the test subject. We believe our tool is suitable to serve the purpose of detecting code clone-related bugs in large software.

Keywords

Software Maintenance
Change Inconsistency
Inconsistent Renaming Bug

Contents

List of Figures

List of Tables

1 Introduction

Code cloning is regarded as one of the major threats to software maintenance. A system containing a large proportion of code clones is more error prone and resistant to change. Despite its negative impacts, code cloning is a common practice in software development even to the professional programmers.

In this chapter, we explain the center problem brought by code cloning that we would like to address, followed by our research goal and major contributions. At the end of this chapter, we provide an outline for the subsequent chapters.

1.1 Problem Definition

Recent studies show that large software systems contain a significant amount of identical or similar code, commonly known as code clones. For instance, [2] found that 19% of the entire source code of X Window system (714,479 lines of code) was identified as duplicates. Another study [16] found that 12% of the Linux file system (279,118 lines of code) involved in code cloning activity.

In many cases, code clones are introduced to the program through copy-and-paste activity. While this practice can greatly reduce programming effort by reusing code in fast and easy way, it is prone to create bugs. During code cloning, the pasted code will normally go through some modifications in order to implement the desired functionality correctly. One of such modifications is to rename all instances of an identifier in pasted code. When this modification is done manually, there is possibility that the renaming is not completely applied to all relevant instances by mistake, therefore introducing unintended inconsistency which is considered as bug to the system. According to [6, 20, 12], abovementioned unintended inconsistency is actually happened in production systems.

Although manual inspection is an effective method to find out bugs in program, it would be too time-consuming and impractical for large software system. Jablonski et al. [14] proposed a tool called CReN, which is implemented as a plug-in to the integrated development environment (IDE) to track copy-and-paste activities and to ensure consistent renaming of identifiers. However, CReN is only applicable when coding a new program or to add new code to an existing program. As such, there is clearly a pressing need to have a tool that can detect bugs introduced by inconsistent renaming of identifiers in production systems.

1.2 Research Goal and Contributions

The detection of code clones has been an emergence research topic since the presence of duplicated code was identified as one of the factors that decreases software maintainability. In parallel with the detection technique, many studies on visualization and refactoring support of code clones have been done. These works improve the understanding of code clones in large software and assist to eliminate code clones by extracting them into a separate new procedure.

On the other hand, research on technique that supports the detection of code clone-related bugs is relatively little. Given the extensive use of copy-and-paste operations and their tendency to introduce bugs as described in section 1.1, an aid to the detection of bugs especially in production systems is indeed necessary.

In this research, we propose a method to detect bugs caused by inconsistent change of identifiers. When it comes to a software with millions lines of code, developer always clueless about the starting point to find out these bugs. The proposed method address this problem by creating a list of possible bugs of the system. By reviewing a small portion of source code, one is able to identify real bugs which may have lurked in the software for long time.

In summary, the research goal is to provide support and to alleviate the effort on detection of bugs introduce by change inconsistencies for large software system.

List of Contributions

The work presented in this thesis contains the following contributions:

- We propose an approach to detect bugs introduce by inconsistent renaming of identifiers and developed a tool based on this approach. The tool comes with an interface that provides necessary information presented in decent way to ease the process of reviewing bug candidates.

- We present a case study on large open source software. We checked the potential bugs detected by our tool against the change log and we found that many of them has been corrected in later version of the subject software. Our approach is proven to reveal bugs.

1.3 Structure of the Thesis

Chapter 2 presents the background knowledge and technologies relevant to our research. We give a clear definition of important terms related to code clones used in this thesis. We explain the reasons why code clones exist in software. Also, an overview on some of the clone detection tools available and their comparison is provided.

Chapter 3 explains our approach to solve the defined problem, which is to detect change inconsistency-induced bugs. The approach consists of three major steps, and we give the details of what each step does.

Chapter 4 discusses the technologies used in order to implement our bug detection tool. We offer the reasons why we have chosen these technologies in our implementation.

Chapter 5 presents the case study on Linux kernel version 2.6.6. We show the outcome that we obtained through experiments, explain our validation method on the experiment results and offer a discussion regarding the results.

Chapter 6 gives the limitation of our tool and discusses some related work.

Finally in Chapter 7, we summarize our work and give an overlook on possibilities for future work.

2 State of the Clone

In this chapter, we give the definitions of some terminology used in code clone detection, followed by the reasons why code clones exist in software. We also provide a brief introduction to some tools available for code clone detection. Finally, we present a comparison on these tools.

2.1 Definitions

The literature currently gives no consistent definition on terminology of code clone-related technologies. Therefore, we defines the terminology and expressions used in the context of this thesis as below.

> ***Definition 1 (Code Clone)*** A code clone is a piece of source code that is identical or similar to another.
>
> ***Definition 2 (Clone Relation)*** A clone relation is defined as an equivalence relation (reflexive, transitive, and symmetric relations) on code fragments. It holds between two code fragments if and only if they are in same token sequences. [15]
>
> ***Definition 3 (Clone Pair)*** A clone pair is a pair of code fragments if the clone relation holds between them.
>
> ***Definition 4 (Clone Set)*** A clone set is a maximal set of code fragments where the clone relation holds between any tuple of code fragments in the set.

The above definitions are further explained using the example illustrated in Figure 1. In this example, there are two source files with five code fragments, f_1 to f_5. f_2 holds the clone relation with f_4, while f_1,f_3, and f_5 hold the clone relation with each other. Therefore, four clone pairs and two clone sets exist.

clone pair : (f_2, f_4), (f_1, f_3), (f_1, f_5),(f_3, f_5)

clone set : $\{f_1, f_3, f_5\}$, $\{f_2, f_4\}$

2.2 Reasons of Presence of Code Clone

Code clones are actively produced because of various reasons. We summarize some common reasons in the following list.

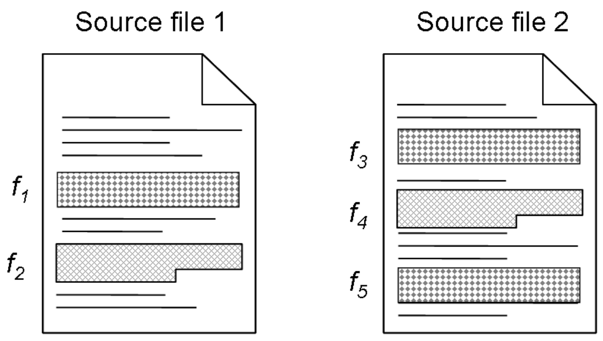


Figure 1: Clone Pair and Clone Set

Copy and Paste
Copy and paste is considered as a primitive reuse mechanism that is commonly practiced by every programmer. Although many methods have been proposed to develop reusable software components, there still exist a lot of ad-hoc reuses through copying and pasting of code.

Lacking Abstraction Mechanism
If the programming language lacks of some abstraction mechanisms such as inheritance, generic types, and parameter passing, programmers will have to implement these as idioms repeatedly. This will lead to possibly small but potentially frequent clones [21].

Stereotyped Process
File open/close operation and database access are typical stereotyped processes while developing a software product. Cordy reported in [7] that there are only limited number of different kinds of financial tasks, and the data structure and programs to carry out these tasks are therefore very similar. Code cloning is a common practice in financial industry.

Performance Consideration
While developing real-time application, if the compiler does not offer to inline the code automatically, this will have to be done by hand. The expanded code becomes code clones. Also, some common operations may have been hand-optimized to achieve best performance. The same existing optimized code is applied whenever the same operation is needed.

Generated Code
Code created by code generators tends to include many code clones since the

tools often use the same template to generate same or similar logic code. Only identifier names of these code clones are different from each other.

2.3 Code Clone Detection Tools

In recent years, several tools have been developed for code clone detection. The tools are based on various techniques, including token-based analysis, line-based analysis, abstract syntax trees, program dependency graphs, and metrics. The following are brief descriptions of some of these tools.

Dup
Dup [1, 2] uses a line-based comparison approach to detect clones. The source code is represented in a sequence of lines. White spaces and comments are eliminated, and identifiers (e.g. function names, variable names, and type) are replaced with a special identifier. The resulting normalized lines are compared and the extraction of matches is performed using suffix-tree algorithm. Dup returns the longest possible fragments of cloned source code. To avoid getting too short, the minimum length of a code clone can be specified. The tool use a dot plot graph to visualize the result and to ease the comprehension of the result.

The extraction of matches in Dup leads to $O(\mathrm{n})$ time complexity, where n is the number of lines in the input. The line-by-line method cannot detect the clone if the line structure is modified. In free-format language such as C, C++, and Java, line breaks in source code have no semantic meaning. They are often placed and relocated based on programmer's preference.

Duploc
Duploc [8] is a clone detection tool that applying a language independent approach. In this approach, the code is only slightly transformed using string manipulation operations. Comments and white spaces are removed to get a condensed form of the line. Each transformed source line is then compared to every other source line by string-matching. The result is a boolean true if two lines match. The value is stored in a matrix where the coordinate is determined by the line index.

Clone pairs are displayed in the form of a scatter plot directly derived from the matrix created during source line comparison. A pattern matcher is run over the matrix to catch cloned code that was changed inside one line of code.

The necessary comparison leads to a complexity of $\Omega(\mathrm{n}^2)$ for input size of n lines which is definitely too expensive. In order to reduce the computational complexity and therefore increase the scalability of tool, a hash function for line is introduced.

CloneDR

CloneDR [3] parses the source code and produces an abstract syntax tree (AST) as first step. After that, three algorithms are applied.

The first algorithm is to detect duplicated subtrees. In order to find matching candidates with some variations, the subtrees are not checked for equality but rather for similarity. For this reason, the subtrees are categorized using a hash function into buckets and only the subtrees in the same bucket are compared with each other.

The second algorithm is to detect clone involving certain recurring fragments like sequences of declarations and statements. The algorithm returns the longest sequence, thus reduces the number of clones but increases their average size.

The third algorithm looks for combinations of clones that can be generalized by a most general unifier.

The computational complexity involved in CloneDR is $O(\mathrm{n})$, where n is the number of subtrees of the source files. In AST approach, it is possible to transform the source trees to regular form but AST-based transformation is generally expensive since it requires full syntax analysis and transformation.

CCFinder

CCFinder [15] is a clone detection tool that applicable to large-scaled software systems with affordable computational complexity. The tool has relatively small language dependent parts which make it adoptable to many languages. The clone detection process of CCFinder consists of the following steps.

1. **Lexical Analysis:** Each line of source files is divided into tokens corresponding to lexical rule of the programming language. The tokens of all source files are concatenated into a single token sequence. White spaces and comments are removed in this step. These suppressed characters are kept in memory for later formatting of the result.
2. **Transformation:** The token sequence is transformed based on transformation rules that aimed at regularization of identifiers and identification of structures. Details of these transformation rules can be found in [15].
3. **Match Detection:** From all the substrings on the transformed token sequence, equivalent pairs with minimum length defined by user are detected as clone pairs.

4. **Formating:** Each location of detected clone pairs is converted into line and column numbers on the original source files.

2.4 Comparison on Code Clone Detection Tools

In this section, we will briefly compare some attributes of the clone detection tools presented previously. There are literatures [4, 5] that provide comprehensive comparison and evaluation on them.

The literatures generally agree that there is no clear winner among code clone detection tools because all tools have their strength and weaknesses. Therefore, they are suitable for different tasks and contexts. We are particularly interested to compare the following attributes among the tools. The comparison is mainly refer to [4, 9].

Approach
Both Dup and CCFinder use a token comparison approach. However, the granularity of Dup is on line level. Duploc compares whole lines to each other textually. CloneDR partitions subtrees of the abstract syntax tree of a program and compares subtrees in the same partition through tree matching.

Scalability
Bellon et al. evaluated all four tools in their work. CloneDR, Dup and CCFinder were able to analyze their largest subject program, postgresql but Duploc failed to do it. Postgresql consists of 235K SLOC written in C.

Recall
Dup, CCFinder and Duploc have higher recall as compared to CloneDR. In other words, tools based on token and text have higher recall.

Precision
CloneDR gives higher precision as compared to Dup, CCFinder and Duploc. This is the opposite case of recall.

Table 1: Comparison on Code Clone Detection Tools

Tool	Approach	Scalability	Recall	Precision	Performance	
					Speed	Memory Consumption
Dup	Token	High	High	Low	V. High	Low
Duploc	Text	Low	High	Low	Low	High
CloneDR	AST	High	V. Low	V. High	Low	High
CCFinder	Token	V. High	High	Low	High	Low

Performance
For both execution speed and memory consumption, Dup and CCFinder are superior to CloneDR and Duploc.

3 Detection of Fault in Code Clones

This chapter introduces an approach with the goal of solving the problem defined in Section 1.1. Figure 2 gives an overview of the approach.

The initial input to the approach is source files and the final output will be a bug candidate list which gives the details such as location of potential bugs. The approach is generally divided into clone detection phase and inconsistency detection phase. The inconsistency detection phase can be further divided into 3 steps, namely lexical analysis, mapping analysis and result filtering.

First, the clone detection tool detects code clones in input source files and produce a clone position file that contains information about the location of detected clones. Then, the code fragments are retrieved from source files based on the clone position file and they are passed to lexical analyzer for tokenization. Next, token mapper performs mapping of identifiers on each tokenized clone pairs. The mapping aims to detect change inconsistencies in clone pairs. The inconsistencies detected will finally go though some metric calculations in order to filter out insignificant inconsistencies which have low possibility to be a bug.

As illustrated in Figure 2, the approach is a step-by-step process where the later step in the approach is to further process the data obtained from the previous step. We use an existing tool to perform the clone detection task. However, the approach is designed to be easily adaptable to other available clone detection tools, especially those token-based tools. It is realized through the implementation of clone information processor module, which serve as the interface between output file of clone detection tool and subsequent steps in our approach. We will describe how this work in detail in Chapter 4.

The three steps in inconsistency detection phase will be explained in following sections.

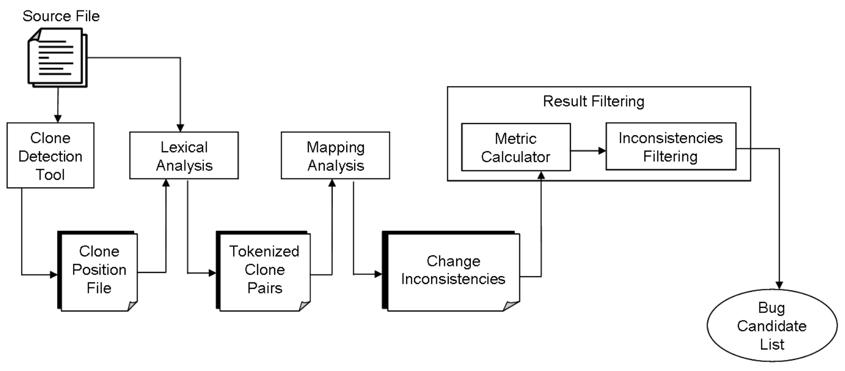


Figure 2: Fault Detection Approach Overview

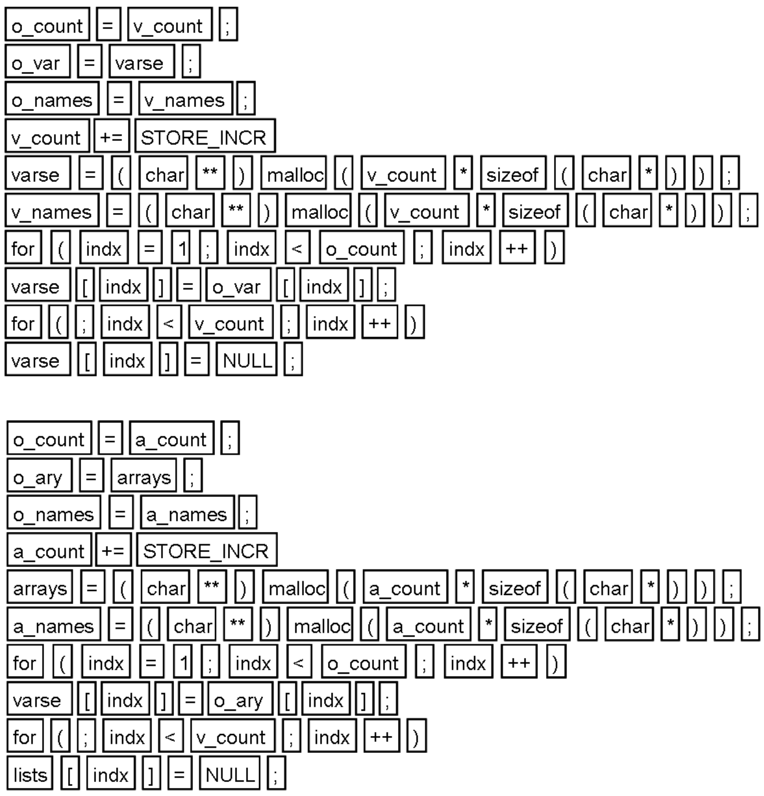
```
127:  o_count = v_count;
128:  o_var = varse;
129:  o_names = v_names;
130:
131:  v_count += STORE_INCR;
132:  varse = (char **) malloc (v_count*sizeof(char *));
133:  v_names = (char **) malloc (v_count*sizeof(char *));
134:
135:  for (indx = 3; indx < o_count; indx++)
135:    varse[indx] = o_var[indx];
137:
138:  for (; indx < v_count; indx++)
139:    varse[indx] = NULL;
      ......

161:  o_count = a_count;
162:  o_ary = arrays;
163:  o_names = a_names;
164:
165:  a_count += STORE_INCR;
166:  arrays = (char **) malloc (a_count*sizeof(char *));
167:  a_names = (char **) malloc (a_count*sizeof(char *));
168:
169:  for (indx = 1; indx < o_count; indx++)
170:    varse[indx] = o_ary[indx];
171:
172:  for (; indx < v_count; indx++)
173:    lists[indx] = NULL;
```

Figure 3: Example of Code Clones

3.1 Lexical Analysis

The general goal of lexical analysis in our approach is to transform the code clones detected in clone detection phase into structure that afterward fed into the mapping analysis step. Code clones that exist in target software are divided into tokens according to lexical rules of the programming language and each of these code clones will form a token sequence. While lexical analyzer scans through the code clones, it tokenizes them and identifies tokens made up by identifier. The position of these tokens is stored because they form the comparison unit in mapping analysis. Comments and white spaces are eliminated, giving a normalized token sequence.

In a clone pair, if one code fragment is an exact copy without modification of another, or only variable, type or function identifiers were changed, the resulting normalized token sequences will have the same number of tokens with identifier at the same position (i.e. same token index). Tokenized clone pairs that fulfill this criteria will be selected for further processing in next step.



```
o_count = v_count ;
o_var = varse ;
o_names = v_names ;
v_count += STORE_INCR
varse = ( char ** ) malloc ( v_count * sizeof ( char * ) ) ;
v_names = ( char ** ) malloc ( v_count * sizeof ( char * ) ) ;
for ( indx = 1 ; indx < o_count ; indx ++ )
varse [ indx ] = o_var [ indx ] ;
for ( ; indx < v_count ; indx ++ )
varse [ indx ] = NULL ;

o_count = a_count ;
o_ary = arrays ;
o_names = a_names ;
a_count += STORE_INCR
arrays = ( char ** ) malloc ( a_count * sizeof ( char * ) ) ;
a_names = ( char ** ) malloc ( a_count * sizeof ( char * ) ) ;
for ( indx = 1 ; indx < o_count ; indx ++ )
varse [ indx ] = o_ary [ indx ] ;
for ( ; indx < v_count ; indx ++ )
lists [ indx ] = NULL ;
```

Figure 4: Tokenization of Code Clone

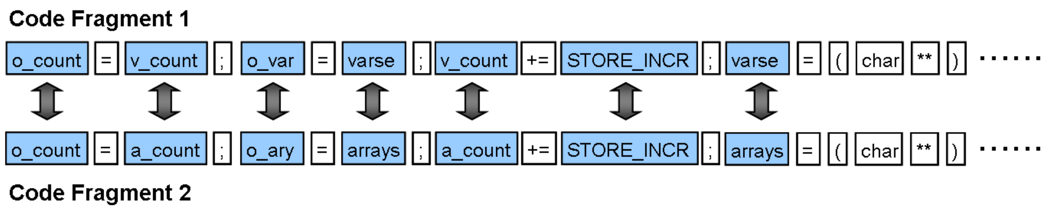


Figure 5: Mapping of Identifiers in Clone Pair

In Figure 3, code fragments consist of line 92-99 and line 111-118 are detected as a pair of code clones. Each of these code fragments will be tokenized as shown in Figure 4, giving two token sequences. Since these two code fragments are syntactically identical, the resulting token sequence will have the same number of token with identifier at the same position in each normalized token sequence.

3.2 Mapping Analysis

Mapping analysis aims to identify the identifier name inconsistency between a clone pair. Each identifier in a code fragment is mapped to the identifier at the same position in another code fragment within a pair of code clones (refer to Figure 5). The mapping is done based on the identifier's position detected in previous step. Therefore, it is important to ensure that only the clone pairs with same sequence are mapped.

If the instances of an identifier in one code fragment are being renamed to more than one identifier names, or not all instances of an identifier is renamed, inconsistency are said to be occurred. For each unique identifier in a code fragment, the mapping is performed and its result is stored.

For the code fragments in a clone set, mapping is performed to all possible combination of clone pairs. Most of the clone detection tools are able to find out clone pairs with differences in identifier names including variables, types, methods, etc but give no information on clone history. In other words, we cannot determine a code fragment is duplicated from which code fragment in a clone set. Therefore, we need to perform the mapping on every possible combination of clone pairs in a clone set.

Table 2 shows one of such mapping results using the example shown in Figure 3. All identifiers existed in code fragment 1 (line 127-139) are renamed consistently in code fragment 2 (line 161-173) except *v_ count* and *varse*. There is one of the instances of *v_count* (highlighted in Figure 3) is left unchanged in fragment 2. On the other hand, instances of *varse* (italicized in Figure 3) are changed to 2 different identifier names, *arrays* and *lists*, while one is left unchanged. Inconsistencies are said to be occurred in this 2 cases.

Table 2: Identifier Mapping Result for Clone Pair in Figure 3

Identifiers in Code Fragment 1 (line 127-139)	Identifiers in Code Fragment 2 (line 161-173)	Occurrence
malloc	malloc	2
indx	indx	8
o_count	o_count	2
o_name	o_name	1
o_var	o_ary	2
v_count	a_count	4
	v_count	1
v_name	a_name	2
varse	arrays	2
	lists	1
	varse	1
STORE_INCR	STORE_INCR	1

3.3 Result Filtering

Inconsistencies detected from the identifier mapping result are not necessary the case of unintended inconsistency resulted from programmer's careless mistakes. In other words, these inconsistencies are not always bugs. Therefore, they need to go through some filtering algorithm in order to produce a list of potential bugs with less false positives. In our approach, we use 2 metrics to serve the filtering purpose.

1. ***UnchangedRatio***
   The first metric is *UnchangedRatio* as proposed in [20]. It is defined as

$$UnchangedRatio(v) = \frac{NumOfUnchangedID(v)}{TotalNumOfID(v)}$$

   For identifier v, *NumOfUnchangedID* is the number of occurrences of v in code fragment 2, f_2 where its name is   remain unchanged as compared to identifier at the same position in code fragment 1, f_1. *TotalNumOfID* is the total number of occurrences of v in f_1. f_1 and f_2 form a clone pair.

   When *UnchangedRatio* equals to 0, it means all instances of an identifier are being renamed. In contrast, when *UnchangedRatio* equals to 1, it means all instances of an identifier are remained in the same name. It is believed that if an identifier is renamed in most of its instances and only a few of its instances are not renamed, there is high possibility that developer forgets to change them. Therefore, the smaller the value of *UnchangedRatio* (except zero), the higher the possibility of the not renamed identifier to be a bug. Inconsistencies with the value of *UnchangedRatio* below the threshold that we set will be reported as bug candidate. Since the original code fragment in a clone pair cannot be determined, *UnchangedRatio* has to be calculated in both directions.

Referring to Table 2, 4 instances of *v_count* are changed to *a_count* in Fragment 2 while one instance is remained in the same name, result in *UnchangedRatio(v_count)* = 0.2. Similarly, 3 instances of *varse* are changed to *arrays* and *lists* while one instance is left unchanged, result in *UnchangedRatio(varse)* = 0.25.

2. ***Conflict***
   UnchangedRatio does not provide any information if an identifier is changed to multiple names. Therefore, we suggest another metric called *Conflict* to complement *UnchangedRatio*. When an identifier is changed to multiple names, *Conflict* is set to true. In Table 2, *varse* is one of such examples. In this case, the possibility that the instances of an identifier are changed to multiple names intentionally to implement the functionality is high. We filter out this kind of inconsistencies even though their *UnchangedRatio* are below threshold value.

4 Implementation of Fault Detection Tool

We have implemented a tool based on the proposed method. The system takes raw source files as input and produces a bug candidate list displayed on a graphical interface as output. This chapter gives the details related to our implementation.

4.1 Overview

The system is implemented using Java. Currently the system can handle programs written in C language and Java. We have carefully modularize the system in order to cater the future enhancement especially in adapting to different clone detection tools and handling various programming languages.

We will explain the details of code clone detection in our current implementation and our consideration to adapt to different clone detection tools in section 4.2. Section 4.3 explains the techniques that we used to filter the code clones detected by clone detection tool applied in our implementation. In section 4.4, the implementation of lexical analyzer and the consideration of handling more programming languages are presented. Following that, we explain how we visualize the bug candidates in order to ease the bug inspection task in section 4.5.

4.2 Code Clone Detection

Code clone detection is an important step since code clone is the initial input to our approach. Therefore, the effectiveness of clone detection affects the result. There are plenty of tools that available for clone detection [15][2][8]. Among them, we have chosen CCFinder to perform the clone detection task in our approach. CCFinder is a token-based clone detection tool which offer good scalability and execution time to cope with large software. The comparison between CCFinder and other tools is presented in section 2.4.

In order to process the output file of CCFinder that contains the clone position information, we implemented a module called clone information processor to interpret this information. This makes our approach having the flexibility to work with different clone detection tools especially those are token-based. By making modification only to clone information processor module, our approach will be able to handle the output file of different clone detection tools.

Figure 6 shows an example of CCFinder output file. The interpretation of tags used in the output file is as below.

#begin{file description} ...... **#end{file description}**
Each line within the tags indicates an input source file where each of these files is given an unique identifier.

#begin{clone} ...... **#end{clone}**
This portion gives the information about code clones detected in input source files. There are 2 presentation styles which are corresponding to pair-wise and class-wise format of CCFinder. In our implementation, we have chosen class-wise format, same to the example shown in Figure 6. By choosing this format, the code clones are grouped into clone set.

#begin{set} ...... **#end{set}**
Each line enclosed by these tags represents a code fragment. The numbers on the line is interpreted as

$$\underset{\text{I}}{\underline{0.0}}\ \underset{\text{II}}{\underline{15}},\underset{\text{III}}{\underline{1}},6\ \underset{\text{IV}}{\underline{21}},\underset{\text{V}}{\underline{32}},41\ 8$$

I:	File ID
II:	Start line
III:	Start column
IV:	End line
V:	End column

For above example, the code fragment is in source file (0.0), begin at line 15 column 1 and end at line 21 column 32.

The clone information processor has been implemented to extract the information from this output file. It will identify the tags and interpret the line resided in between the tags according to the format explained above. The information is stored and lexical analyzer will extract the code clones from source files based on this information in order to perform the tokenization of code clones.

4.3 Code Clone Filtering

One of the factors that affects the output of CCFinder is the minimum length of code clone. The higher the number, the less code clones will be found by CCFinder. In our experiments, we set the minimum length of a code clone to 30 tokens. This number was used in numerous previous researches [17][18] on CCFinder and gave positive results.

CCFinder identifies a large amount of code clones, which some of them are deemed insignificant to our bug detection method. Therefore, it is important to filter the raw output of CCFinder and select only good candidate clones as the input to our proposed method. We applied the following two techniques to filter code clones detected by CCFinder and conducted experiments (details in section 5.2) to evaluate their effectiveness.

```
#version: ccfinder 7.3.2
#format: classwise
#langspec: C
#option: –b 30
#option: –e char
#option: –k 30
#option: –r abdfikmnpstuv
#option: –c wfg
#option: –y
#begin{file description}
0.0 36 102 C:\source_for_test_system\linux–2.6.6\arch\m68k\amiga\amiga_ksyms.c
0.1 520 1077 C:\source_for_test_system\linux–2.6.6\arch\m68k\amiga\amiints.c
0.2 113 188 C:\source_for_test_system\linux–2.6.6\arch\m68k\amiga\amisound.c
0.3 133 268 C:\source_for_test_system\linux–2.6.6\arch\m68k\amiga\chipram.c
0.4 180 363 C:\source_for_test_system\linux–2.6.6\arch\m68k\amiga\cia.c
0.5 1007 2361 C:\source_for_test_system\linux–2.6.6\arch\m68k\amiga\config.c
                  :
                  :
#end{file description}
#begin{syntax error}
#end{syntax error}
#begin{clone}
#begin{set}
0.0 15,1,6 21,32,41 8
0.0 16,1,11 22,33,46 3
  :
  :
#end{set}
#begin{set}
0.0 15,1,6 22,33,46 8
0.0 26,1,61 35,39,101 0
  :
  :
#end{set}
  :
  :
#end{clone}
```

Figure 6: Example of CCFinfer Output file

1. **Removing Highly Repeated Code Sequence Clones**

   There is a large portion of code clones detected by CCFinder consists of consecutive variable declarations and consecutive method invocations. These kinds of clones are mainly due to the structure of programming language and many of them are stereotyped code which is very stable. We filter out these kinds of clones using a metric called *RNR(S)* that represent the ratio of non-repeated code sequence in clone set S [10].

   Let clone set S consists of n fragments, $f_1, f_2, ..., f_n$. $LOS_{whole}(f_i)$ represents the Length Of whole Sequence of fragment f_i, and $LOS_{repeated}(f_i)$ represents the Length Of repeated Sequence of fragment f_i, *RNR(S)* is defined as

   $$RNR(S) = 1 - \frac{\sum_{i=1}^{n} LOS_{repeated}(f_i)}{\sum_{i=1}^{n} LOS_{whole}(f_i)}$$

   Repeated code sequence is the repetition of its adjacent code sequence and non-repeated code sequence is the other parts. In our experiment, we set the *RNR* value to 0.5, filtering out clone set with the value less than 0.5 from the input to our method. This is the value suggested in [10].

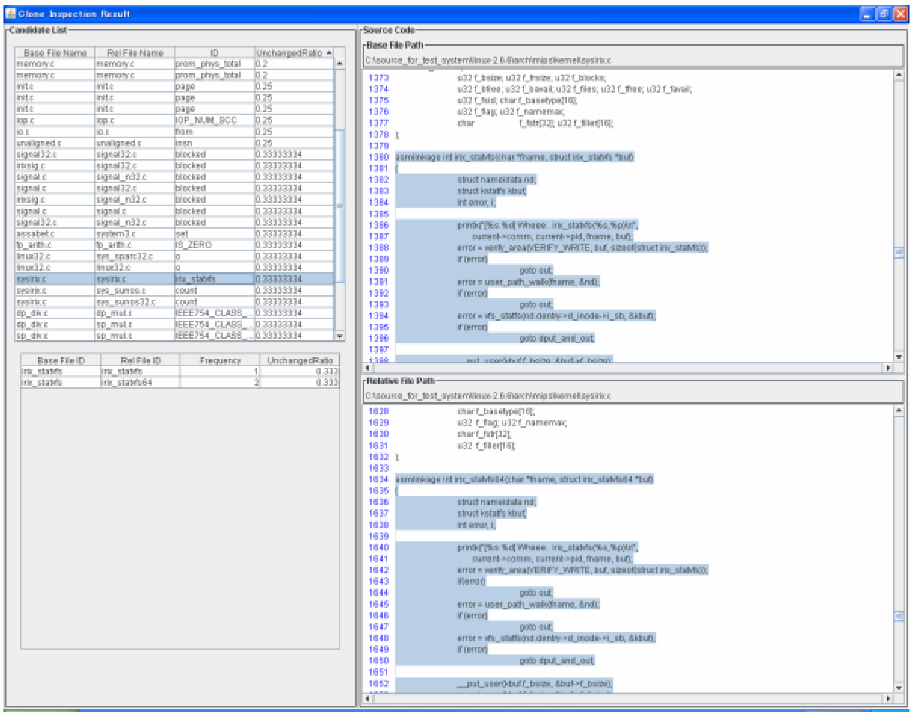
2. **Removing Overlapping Clones**

   Among the clones detected by CCFinder, we discovered a lot of overlapping clones where a portion of code fragments in a clone pair overlap each other. Overlapping clones do not represent the nature of copy-and-paste process and most probably created coincidently. Since our method is based on the assumption of copy-paste-modify mechanism, overlapping clones are deemed insignificant to our method and need to be filtered out.

4.4 Code Clone Tokenization

The lexical analyzer that is responsible to tokenize code clones is implemented using JFLEX [11], a lexical analyzer generator. We need to prepare a lexical specification of our target programming language in JFLEX 's specification syntax. JFLEX provides various options in its specification syntax, including the option to incorporate with user code. Therefore, we have good flexibility to control the behavior of generated lexical analyzer and able to interface with other modules of our tool easily. We include a lexer specification of Java in appendix.

A lexical specification file for JFLEX consists of three parts divided by a single line starting with %%:



Base File Name	Rel File Name	ID	UnchangedRatio
memory.c	memory.c	prom_phys_total	0.2
memory.c	memory.c	prom_phys_total	0.2
init.c	init.c	page	0.25
init.c	init.c	page	0.25
init.c	init.c	page	0.25
iop.c	iop.c	IOP_NUM_SCC	0.25
io.c	io.c	from	0.25
unaligned.c	unaligned.c	insn	0.25
signal32.c	signal32.c	blocked	0.33333334
irixsig.c	signal32.c	blocked	0.33333334
signal.c	signal_n32.c	blocked	0.33333334
signal.c	signal32.c	blocked	0.33333334
irixsig.c	signal_n32.c	blocked	0.33333334
signal.c	signal.c	blocked	0.33333334
signal32.c	signal_n32.c	blocked	0.33333334
sysstat.c	system3.c	set	0.33333334
fp_arith.c	fp_arith.c	IS_ZERO	0.33333334
linux32.c	sys_sparc32.c	o	0.33333334
linux32.c	linux32.c	o	0.33333334
sysirix.c	sysirix.c	irix_statfs	0.33333334
sysirix.c	sys_sunos.c	count	0.33333334
sysirix.c	sys_sunos32.c	count	0.33333334
dp_div.c	dp_mul.c	IEEE754_CLASS_...	0.33333334
dp_div.c	sp_mul.c	IEEE754_CLASS_...	0.33333334
sp_div.c	sp_mul.c	IEEE754_CLASS_...	0.33333334

Base File ID	Rel File ID	Frequency	UnchangedRatio
irix_statfs	irix_statfs	1	0.333
irix_statfs	irix_statfs64	2	0.333

C:\source_for_test_system\linux-2.6.6\arch\mips\kernel\sysirix.c

```
                u32 f_bsize; u32 f_frsize; u32 f_blocks;
                u32 f_bfree; u32 f_bavail; u32 f_files; u32 f_ffree; u32 f_favail;
                u32 f_fsid; char f_basetype[16];
                u32 f_flag; u32 f_namemax;
                char          f_fstr[32]; u32 f_filler[16];
};

asmlinkage int irix_statvfs(char *fname, struct irix_statvfs *buf)
{
                struct nameidata nd;
                struct kstatfs kbuf;
                int error, i;

                printk("[%s:%d] Wheee.. irix_statvfs(%s,%p)\n",
                   current->comm, current->pid, fname, buf);
                error = verify_area(VERIFY_WRITE, buf, sizeof(struct irix_statvfs));
                if (error)
                                goto out;
                error = user_path_walk(fname, &nd);
                if (error)
                                goto out;
                error = vfs_statfs(nd.dentry->d_inode->i_sb, &kbuf);
                if (error)
                                goto dput_and_out;

                  put_user(kbuf.f_bsize, &buf->f_bsize);
```

C:\source_for_test_system\linux-2.6.6\arch\mips\kernel\sysirix.c

```
                char f_basetype[16];
                u32 f_flag; u32 f_namemax;
                char f_fstr[32];
                u32 f_filler[16];
};

asmlinkage int irix_statvfs64(char *fname, struct irix_statvfs64 *buf)
{
                struct nameidata nd;
                struct kstatfs kbuf;
                int error, i;

                printk("[%s:%d] Wheee.. irix_statvfs(%s,%p)\n",
                   current->comm, current->pid, fname, buf);
                error = verify_area(VERIFY_WRITE, buf, sizeof(struct irix_statvfs));
                if(error)
                                goto out;
                error = user_path_walk(fname, &nd);
                if (error)
                                goto out;
                error = vfs_statfs(nd.dentry->d_inode->i_sb, &kbuf);
                if (error)
                                goto dput_and_out;

                __put_user(kbuf.f_bsize, &buf->f_bsize);
```

Figure 7: Snapshot of System's Graphical Interface

User code
%%
Options and declarations
%%
Lexical rules

The first part contains user code that is copied verbatim into the beginning of the source file of the generated lexical analyzer . This is the place to put package declarations and import statements.

The second part contains options to customize the generated lexical analyzer which is specified using JFLEX directives, declarations of lexical states and macro definitions for use in the third section of the lexical specification file.

The third part contains a set of regular expressions and actions that are executed when the scanner matches the associated regular expression.

4.5 Visualization of Fault Candidate

Since our proposed method will produce a list of potential bugs for inspection, it is important to provides necessary information presented in decent way to ease the programmers carry out this job. We have built an graphical interface to browse the bug candidate list. A screen shot of the interface is shown in Figure 7.

The top left frame of the interface is a bug candidate list which can be sorted by the value of *UnchangedRatio*. One can start the inspection by looking at the most suspicious case (i.e identifiers with small value of *UnchangedRatio*). When clicking on an item in bug candidate list, the mapping result of selected identifier will be displayed on the bottom left frame. At the same time, the 2 frames on the right will display the related source files. Code clones where the selected identifier resided in are being highlighted in the source files.

5 Case Study on Linux-2.6.6

In this section, we will describe the details of experiments that we have carried out with the tool implemented based on the proposed method. The main purpose of the case study is to evaluate the ability of our method to find real bugs caused by inconsistent renaming of identifier in production systems.

We have chosen Linux version 2.6.6, which consist of 6,491 files and 4,364,540 lines of code as our experiment subject. We chose Linux because it is a well-known large scale production system. We can test the capability of our tool to cope with large software system, at the same time evaluate the effectiveness of our tool. For the experiment results, we concentrate the discussion on 2 of the largest modules of Linux, arch and drivers.

5.1 Experiment Parameters

There are several parameters that will affect the experiment results. The following describe each of them and the value we set in our experiment.

1. **Threshold Value for *UnchangedRatio***

   As mentioned in Section 3.3, a non-zero or non-one value for *UnchangedRatio* indicates inconsistent change of identifiers. When using this metric to narrow down the bug candidates, we are based on the idea that " the developer intents to changed all instances of an identifier consistently to another name but left out some of them ". As such, smaller value of *UnchangedRatio* (except zero) better reveal a potential bug. In our experiments, we set the value of *UnchangedRatio* to 0.4, same to the value used in [20].

2. ***Conflict* Setting**

   We can specify the degree we tolerate inconsistency by determine how many different names an identifier can be changed to. In our experiment, if an identifier is changed to 2 different names or above, *Conflict* is set to true and it will be filtered out.

5.2 Filtering Code Clones

Table 3 shows the difference of the number of code clones detected by CCFinder from our test subject, Linux 2.6.6 before and after applying *RNR* filtering method. On the other hand, Table 4 gives the number of clone pairs after we took out the overlapped clones. The difference becomes obvious after we applied the *RNR* filtering. The filtering can be set to on or off in the experiment. We have conducted experiments both with and without the filtering and we found that the filtering techniques greatly reduce our investigation effort by reducing the number of bug candidates created by insignificant clones. Therefore, we applied these filtering techniques when running our experiment with Linux 2.6.6.

Table 3: Number of Code Clones in Linux 2.6.6

Module	# Without Filtering	# Filtered with *RNR*
linux-2.6.6/arch	102,539	17,085
linux-2.6.6/drivers	159,764	44,881

Table 4: Number of Clone Pairs in Linux 2.6.6

Module	Before *RNR* Filtering		After *RNR* Filtering	
	With Overlapping Clone	Without Overlapping Clone	With Overlapping Clone	Without Overlapping Clone
linux-2.6.6/arch	16,740,180	14,977,660	23,599	19,273
linux-2.6.6/drivers	8,325,367	7,888,115	60,706	56,260

5.3 Detecting Bugs

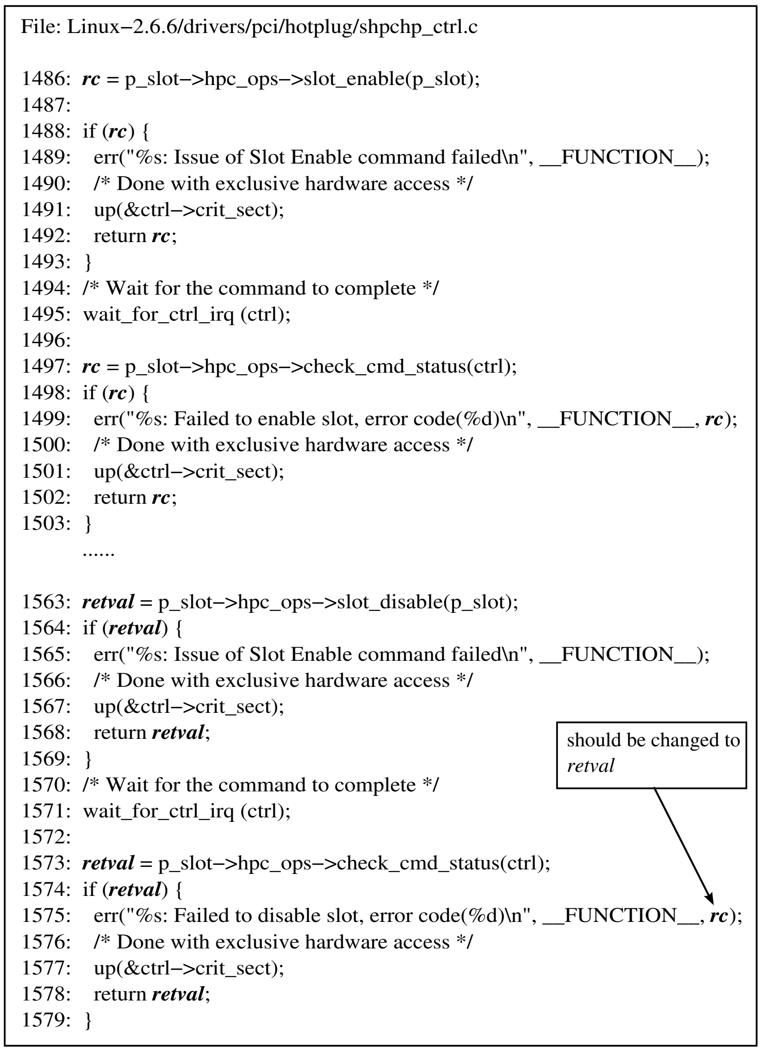
5.3.1 Bug Candidates

The number of bug candidates found in Linux 2.6.6 is shown in Table 5. We found 87 bug candidates in arch module and 120 in drivers module. If a clone pair contains a bug candidate, we call it suspicious clone pair. Adding up the lines of code (LOC) of all suspicious clone pairs gives total LOC of suspicious clones. Sometimes a clone pair can contain more than 1 bug candidate. It that case, we only add once when calculating the total LOC of suspicious clones.

In both arch and drivers modules, the total LOC of suspicious clones occupies less than 0.1% of the total LOC. It gives a rough figure on the total number of lines that we need to review. In reality, we might need to review more code in order to verify a bug candidate, but it serve as a good start especially to detect bugs in a millions-line-of-code software.

Table 5: Number of Bug Candidates in Linux 2.6.6

Module	# File	Total LOC	# Bug Candi-date	Total LOC Suspicous Clone
linux-2.6.6/arch	2,355	724,858	87	1,615
linux-2.6.6/drivers	2,323	2,344,594	120	3,637



```
File: Linux−2.6.6/drivers/pci/hotplug/shpchp_ctrl.c

1486: rc = p_slot−>hpc_ops−>slot_enable(p_slot);
1487:
1488: if (rc) {
1489:   err("%s: Issue of Slot Enable command failed\n", __FUNCTION__);
1490:   /* Done with exclusive hardware access */
1491:   up(&ctrl−>crit_sect);
1492:   return rc;
1493: }
1494: /* Wait for the command to complete */
1495: wait_for_ctrl_irq (ctrl);
1496:
1497: rc = p_slot−>hpc_ops−>check_cmd_status(ctrl);
1498: if (rc) {
1499:   err("%s: Failed to enable slot, error code(%d)\n", __FUNCTION__, rc);
1500:   /* Done with exclusive hardware access */
1501:   up(&ctrl−>crit_sect);
1502:   return rc;
1503: }
      ......

1563: retval = p_slot−>hpc_ops−>slot_disable(p_slot);
1564: if (retval) {
1565:   err("%s: Issue of Slot Enable command failed\n", __FUNCTION__);
1566:   /* Done with exclusive hardware access */
1567:   up(&ctrl−>crit_sect);
1568:   return retval;
1569: }
1570: /* Wait for the command to complete */
1571: wait_for_ctrl_irq (ctrl);
1572:
1573: retval = p_slot−>hpc_ops−>check_cmd_status(ctrl);
1574: if (retval) {
1575:   err("%s: Failed to disable slot, error code(%d)\n", __FUNCTION__, rc);
1576:   /* Done with exclusive hardware access */
1577:   up(&ctrl−>crit_sect);
1578:   return retval;
1579: }
```

Figure 8: Bug Caused by Inconsistent Renaming of Identifier

5.3.2 Bugs Verification

One of the major tasks in our result analysis is to verify the bug candidates detected by our tool. We have inspected the bug candidates in arch and drivers modules. Some of the bug candidates are in high possibility. We checked them against Linux change log and later version of Linux. As a result, we were able to verify some of them to be the genuine bugs.

For arch module, we have verified all 87 bug candidates listed by our tool. Out of 87 bug candidates, 3 of them were rectified by Linux developers and the changes were listed in Linux change log. We also found that 5 candidates are in high possibility to be a real bug.

Table 6: Genuine Bugs Found in Linux 2.6.6

File Path	File Line	Identifier
../arch/m68k/mac/iop.c	264	IOP_NUM_SCC
../arch/sparc/prom/memory.c	159	prom_phys_total
../arch/sparc64/prom/memory.c	117	prom_phys_total
../drivers/pci/hotplug/shpchp_ctrl.c	1575	rc

Figure 8 gives an example of bug that is found in drivers module of Linux version 2.6.6. Code fragments consist of line 1486-1503 and line 1563-1579 are detected as a pair of code clones. This clone pair could be the result of copy-and-paste process. All variables named *rc* in pasted code fragment (line 1563-1579) have to be changed to *retval* but the one in line 1575 was left out. In consequence, the system displays wrong error code. This bug cannot be detected by compiler since the variable *rc* in pasted code fragment is still within the valid scope, thus produce no warning or syntax error.

Table 6 gives some instances of verified bug with the file path, line number and identifier that is renamed inconsistently.

6 Discussion and Related Work

6.1 Limitation of Tool

Our tool currently cannot handle gapped clone that is created when a copy-and-paste code fragment gone through modifications such as insertion and deletion of statements. Gapped clones should be inspected as well since they also have the possibility of containing inconsistent renaming bugs. The greatest barrier for our current implementation to handle gapped clones is on the mapping analysis. Our method still lack of comprehensive algorithm to correctly map the identifiers in gapped clone. This gives a space for our tool to be improved.

6.2 Related Work

Li et al. [20] have developed a tool called CP-Miner to detect copy-and-paste related bugs. CP-Miner uses data mining techniques to identify code clones and the bug detection is implemented as part of the tool. Before passing to bug detection process, code clones detected gone through a series of pruning procedures. CP-Miner has the ability to handle clone that is not exactly the same.

Recently, Jiang et al. [12] developed a tool to detect bugs in code clones and their surrounding code, called context. The clone detection component of the tool is based on DECKARD [13], a clone detection tool developed by them. Bug caused by renaming inconsistency is called a type-3 inconsistency in their work. The tool counts the number of unique identifiers in each code fragment within a pair of clones. Different number of unique identifiers is deemed as likely to be a bug.

7 Conclusion

Code cloning is a problem that arises in every software project. When a piece of code is duplicated through copy-and-paste, the process is always followed by modifications to pasted code. One of such modifications is to rename all instances of an identifier in pasted code. As indicated in previous work, it is easy for developers to miss some instances and thus introduce subtle bugs.

In this thesis, we introduced a method to detect bugs caused by inconsistent change of identifiers. We first detect the code clones existed in software and identify the location of identifiers in code clones. Then, the mapping of identifiers within clone pairs is performed in order to find out renaming inconsistency of identifiers. By using 2 predefined metrics, we filter out the detected inconsistencies which have low possibility to be a bug. Finally, the approach gives a list of potential bugs for inspection.

We have conducted experiments using Linux version 2.6.6 in order to evaluate the effectiveness of our proposed method. We have checked the bug candidates against Linux change log and later version of Linux-2.6.6. As a result, we were able to discover bugs on this system. Therefore, we believe that our tool is effective for detecting bugs in large software systems.

In future, we would like to conduct more experiments on production software and perform a comprehensive analysis on the results. The analysis will help us to refine the filtering method in order to precisely filter out more false positives from bug candidate list. Also, we would like to improve our tool to handle more clone patterns especially the gapped clones. This will involve the refinement in mapping analysis.

Acknowledgments

First and foremost, I would like to thank Professor Katsuro Inoue for giving me the opportunity to work with him. He provided advices and important direction to my thesis work.

Special thanks to Associate Professor Makoto Matsushita and Assistant Professor Yasuhiro Hayase for their valuable comments and guidance especially in technical matters.

I would like to express my gratitude to all members of Department of Computer Science for their guidance.

Thanks are also due to many friends in Department of Computer Science, especially students in Inoue Laboratory.

References

[1] B.S. Baker, “ A Program for Identifying Duplicated Code, ”*Proceedings of the 24th Symposium of Computer Science and Statistics*, pp. 49-57, March 1992.

[2] B.S. Baker, “ On Finding Duplication and Near-Duplication in Large Software Systems, ”*Proceedings of the Second Working Conference on Reverse Engineering*, pp. 86-95, July 1995.

[3] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier, “ Clone Detection Using Abstract Syntax Trees, ”*Proceedings of the IEEE International Conference on Software Maintenance*, pp. 368-377, March 1998.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, “ Comparison and Evaluation of Clone Detection Tools, ”*IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577-591, September 2007.

[5] E. Burd, and J. Bailey,“ Evaluating Clone Detection Tools for Use during Preventative Maintenance, ”*Proceedings of the second IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 36-43, October 2002.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “ An Empirical Study of Operating Systems Errors, ”*Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp. 73-88, 2001.

[7] J.R. Cordy, “ Comprehending Reality - Practical Challenges to Software Maintenance Automation, ”*Proceedings of IEEE 11th International Workshop on Program Comprehension*, pp. 196-206, May 2003.

[8] S. Ducasse, M. Reiger, and S. Demeyer, “ A Language Independent Approach for Detecting Duplicated Code, ”*Proceedings of the IEEE International Conference on Software Maintenance*, pp. 109-118, August 1999.

[9] R. Geiger, *Evolution Impact of Code Clones: Identification of Structural and Change Smells based on Code Clones*, Diploma thesis, University of Zurich, 2005.

[10] Y. Higo, T. Kamiya, S. Kusumoto and K. Inoue,“ Mehtod and Implementation for Investigating Code Clones in a Software System, ”*Information and Software Technology*, vol. 49, pp. 985-998, September 2007.

[11] G. Klein, *JFlex User ’s Manual*, November 2004.
Available at `http://jflex.de/manual.html`.

[12] L. Jiang, Z. Su, and E.Chiu, “ Context-Based Detection of Clone-Related Bugs, ” *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium of the Foundations of Software Engineering*, pp. 55-64, September 2007.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “ DECKARD: Scalable and Accurate Tree-based Detection of Code Clones, ”*Proceedings of 29th International Conference on Software Engineering*, pp. 96-105, May 2007.

[14] P. Jablonski, and D. Hou, “ CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE, ”*Proceedings of the OOPSLA Workshop on Eclipse Technology Exchange*, pp. 16-20, October 2007.

[15] T. Kamiya, S. Kusumoto, and K. Inoue, “ CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, ”*IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654-670, July 2002.

[16] C. Kapser and M.W. Godfrey, “ Toward a Taxonomy of Clones in Source Code: A Case Study, ”*Evolution of Large-scale Industrial Software Applications*, September 2003.

[17] C. Kapser, and M. Godfrey, “ Supporting the Analysis of Clones in Software Systems: A Case Study, ”*Journal of Software Maintenance and Evolution: Research and Practice*, Vol.18, pp. 61-82, 2006.

[18] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “ An Empirical Study of Code Clone Genealogies, ”*Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium Foundation of Software Engineering*, pp. 187-196, Sept. 2005.

[19] J. Krinke, “ A Study of Consistent and Inconsistent Changes to Code Clones, ”*Proceedings of the 14th Working Conference on Reverse Engineering*, pp. 170-178, July 2007.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou,“ CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, ”*IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176-192, March 2006.

[21] M. Rieger, *Effective Clone Detection Without Language Barriers*, PhD thesis, University of Berne, 2005.

Appendix

A. JFLEX Specification for Java

JFLEX Specification for Java

```
/* Java language lexer specification */
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;
import java.util.LinkedList;
%%
%public
%class JavaScanner
%function scanCode
%type List
%eofclose
%unicode
%line
%column
%init{
  this.tokenList = new LinkedList();
  this.identifierPosList = new ArrayList<Integer> ();
%init}
%{
  StringBuffer string = new StringBuffer();
  private List tokenList;
  private List<Integer> identifierPosList;
  public List<Integer> getIdentifierPosList(){
  return Collections.unmodifiableList(identifierPosList);
  }
 /**
   * assumes correct representation of a long value for
   * specified radix in scanner buffer from <code>start</code>
   * to <code>end</code>
   */
  private long parseLong(int start, int end, int radix) {
    long result = 0;
    long digit;
    for (int i = start; i < end; i++) {
      digit  = Character.digit(yycharat(i),radix);
      result*= radix;
      result+= digit;
    }
    return result;
  }
%}
/* main character classes */
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace = {LineTerminator} | [ \t\f]
/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} |
          {DocumentationComment}
TraditionalComment = "/*" [^*] ~"*/" | "/*" "*"+ "/"
EndOfLineComment = "//" {InputCharacter}* {LineTerminator}?
DocumentationComment = "/*" "*"+ [^/*] ~"*/"
/* identifiers */
Identifier = [:jletter:][:jletterdigit:]*
```

```
/* integer literals */
DecIntegerLiteral = 0 | [1-9][0-9]*
DecLongLiteral    = {DecIntegerLiteral} [lL]

HexIntegerLiteral = 0 [xX] 0* {HexDigit} {1,8}
HexLongLiteral    = 0 [xX] 0* {HexDigit} {1,16} [lL]
HexDigit          = [0-9a-fA-F]

OctIntegerLiteral = 0+ [1-3]? {OctDigit} {1,15}
OctLongLiteral    = 0+ 1? {OctDigit} {1,21} [lL]
OctDigit          = [0-7]

/* floating point literals */
FloatLiteral  = ({FLit1}|{FLit2}|{FLit3}) {Exponent}? [fF]
DoubleLiteral = ({FLit1}|{FLit2}|{FLit3}) {Exponent}?

FLit1    = [0-9]+ \. [0-9]*
FLit2    = \. [0-9]+
FLit3    = [0-9]+
Exponent = [eE] [+-]? [0-9]+

/* string and character literals */
StringCharacter = [^\r\n\"\\]
SingleCharacter = [^\r\n\'\\]

%state STRING, CHARLITERAL

%%

<YYINITIAL> {

  /* keywords */
  "abstract"                  { tokenList.add("ABSTRACT"); }
  "assert"                    { tokenList.add("ASSERT"); }
  "boolean"                   { tokenList.add("BOOLEAN"); }
  "break"                     { tokenList.add("BREAK"); }
  "byte"                      { tokenList.add("BYTE"); }
  "case"                      { tokenList.add("CASE"); }
  "catch"                     { tokenList.add("CATCH"); }
  "char"                      { tokenList.add("CHAR"); }
  "class"                     { tokenList.add("CLASS"); }
  "const"                     { tokenList.add("CONST"); }
  "continue"                  { tokenList.add("CONTINUE"); }
  "default"                   { tokenList.add("DEFAULT"); }
  "do"                        { tokenList.add("DO"); }
  "double"                    { tokenList.add("DOUBLE"); }
  "else"                      { tokenList.add("ELSE"); }
  "extends"                   { tokenList.add("EXTENDS"); }
  "final"                     { tokenList.add("FINAL"); }
  "finally"                   { tokenList.add("FINALLY"); }
  "float"                     { tokenList.add("FLOAT"); }
  "for"                       { tokenList.add("FOR"); }
  "goto"                      { tokenList.add("GOTO"); }
  "if"                        { tokenList.add("IF"); }
  "implements"                { tokenList.add("IMPLEMENTS"); }
  "import"                    { tokenList.add("IMPORT"); }
  "instanceof"                { tokenList.add("INSTANCEOF"); }
  "int"                       { tokenList.add("INT"); }
  "interface"                 { tokenList.add("INTERFACE"); }
  "long"                      { tokenList.add("LONG"); }
  "native"                    { tokenList.add("NATIVE"); }
  "new"                       { tokenList.add("NEW"); }
  "package"                   { tokenList.add("PACKAGE"); }
  "private"                   { tokenList.add("PRIVATE"); }
  "protected"                 { tokenList.add("PROTECTED"); }
  "public"                    { tokenList.add("PUBLIC"); }
  "return"                    { tokenList.add("RETURN"); }
  "short"                     { tokenList.add("SHORT"); }
  "static"                    { tokenList.add("STATIC"); }
```

```
"strictfp"                    { tokenList.add("STRICTFP"); }
"super"                       { tokenList.add("SUPER"); }
"switch"                      { tokenList.add("SWITCH"); }
"synchronized"                { tokenList.add("SYNCHRONIZED"); }
"this"                        { tokenList.add("THIS"); }
"throw"                       { tokenList.add("THROW"); }
"throws"                      { tokenList.add("THROWS"); }
"transient"                   { tokenList.add("TRANSIENT"); }
"try"                         { tokenList.add("TRY"); }
"void"                        { tokenList.add("VOID"); }
"volatile"                    { tokenList.add("VOLATILE"); }
"while"                       { tokenList.add("WHILE"); }

/* boolean literals */
"true"                        { tokenList.add("TRUE"); }
"false"                       { tokenList.add("FALSE"); }

/* null literal */
"null"                        { tokenList.add("NULL"); }


/* separators */
"("                           { tokenList.add("("); }
")"                           { tokenList.add(")"); }
"{"                           { tokenList.add("{"); }
"}"                           { tokenList.add("}"); }
"["                           { tokenList.add("["); }
"]"                           { tokenList.add("]"); }
";"                           { tokenList.add(";"); }
","                           { tokenList.add(","); }
"."                           { tokenList.add("."); }

/* operators */
"="                           { tokenList.add("="); }
">"                           { tokenList.add(">"); }
"<"                           { tokenList.add("<"); }
"!"                           { tokenList.add("!"); }
"~"                           { tokenList.add("~"); }
"?"                           { tokenList.add("?"); }
":"                           { tokenList.add(":"); }
"=="                          { tokenList.add("=="); }
"<="                          { tokenList.add("<="); }
">="                          { tokenList.add(">="); }
"!="                          { tokenList.add("!="); }
"&&"                          { tokenList.add("&&"); }
"||"                          { tokenList.add("||"); }
"++"                          { tokenList.add("++"); }
"--"                          { tokenList.add("--"); }
"+"                           { tokenList.add("+"); }
"-"                           { tokenList.add("-"); }
"*"                           { tokenList.add("*"); }
"/"                           { tokenList.add("/"); }
"&"                           { tokenList.add("&"); }
"|"                           { tokenList.add("|"); }
"^"                           { tokenList.add("^"); }
"%"                           { tokenList.add("%"); }
"<<"                          { tokenList.add("<<"); }
">>"                          { tokenList.add(">>"); }
">>>"                         { tokenList.add(">>>"); }
"+="                          { tokenList.add("+="); }
"-="                          { tokenList.add("-="); }
"*="                          { tokenList.add("*="); }
"/="                          { tokenList.add("/="); }
"&="                          { tokenList.add("&="); }
"|="                          { tokenList.add("|="); }
"^="                          { tokenList.add("^="); }
"%="                          { tokenList.add("%="); }
"<<="                         { tokenList.add("<<="); }
```

```
  ">>="                         { tokenList.add(">>="); }
  ">>>="                        { tokenList.add(">>>="); }

  /* string literal */
  \"                            { yybegin(STRING); string.setLength(0); }

  /* character literal */
  \'                            { yybegin(CHARLITERAL); }

  /* numeric literals */

  {DecIntegerLiteral}           { tokenList.add(new Integer(yytext())); }
  {DecLongLiteral}              { tokenList.add(
                                   new Long(yytext().substring(0,yylength()-1))); }

  {HexIntegerLiteral}           { tokenList.add(
                                   new Integer((int) parseLong(2, yylength(), 16))); }
  {HexLongLiteral}              { tokenList.add(
                                   new Long(parseLong(2, yylength()-1, 16))); }

  {OctIntegerLiteral}           { tokenList.add(
                                   new Integer((int) parseLong(0, yylength(), 8))); }
  {OctLongLiteral}              { tokenList.add(
                                   new Long(parseLong(0, yylength()-1, 8))); }

  {FloatLiteral}                { tokenList.add(
                                   new Float(yytext().substring(0,yylength()-1))); }
  {DoubleLiteral}               { tokenList.add(new Double(yytext())); }
  {DoubleLiteral}[dD]           { tokenList.add(
                                   new Double(yytext().substring(0,yylength()-1))); }

  /* comments */
  {Comment}                     { /* ignore */ }

  /* whitespace */
  {WhiteSpace}                  { /* ignore */ }

  /* identifiers */
  {Identifier}                  { tokenList.add(yytext());
                                   identifierPosList.add(tokenList.size()-1); }
}

<STRING> {
  \"                            { yybegin(YYINITIAL);
                                  tokenList.add(string.toString()); }

  {StringCharacter}+            { string.append( yytext() ); }

  /* escape sequences */
  "\\b"                         { string.append( '\b' ); }
  "\\t"                         { string.append( '\t' ); }
  "\\n"                         { string.append( '\n' ); }
  "\\f"                         { string.append( '\f' ); }
  "\\r"                         { string.append( '\r' ); }
  "\\\""                        { string.append( '\"' ); }
  "\\'"                         { string.append( '\'' ); }
  "\\\\"                        { string.append( '\\' ); }
  \\[0-3]?{OctDigit}?{OctDigit}  { char val = (char) Integer.parseInt(
                                               yytext().substring(1),8);
                                  string.append( val ); }

  /* error cases */
  \\.                           { throw new RuntimeException(
                                     "Illegal escape sequence \""+yytext()+"\""); }
  {LineTerminator}              { throw new RuntimeException(
                                     "Unterminated string at end of line"); }
}

<CHARLITERAL> {
  {SingleCharacter}\'           { yybegin(YYINITIAL);
                                  tokenList.add(new Character(yytext().charAt(0))); }
  /* escape sequences */
  "\\b"\'                       { yybegin(YYINITIAL);
                                  tokenList.add(new Character('\b'));}
  "\\t"\'                       { yybegin(YYINITIAL);
                                  tokenList.add(new Character('\t'));}
  "\\n"\'                       { yybegin(YYINITIAL);
                                  tokenList.add(new Character('\n'));}
  "\\f"\'                       { yybegin(YYINITIAL);
                                  tokenList.add(new Character('\f'));}
  "\\r"\'                       { yybegin(YYINITIAL);
                                  tokenList.add(new Character('\r'));}
  "\\\""\'                      { yybegin(YYINITIAL);
                                  tokenList.add(new Character('\"'));}
  "\\'"\'                       { yybegin(YYINITIAL);
                                  tokenList.add(new Character('\''));}
  "\\\\"\'                      { yybegin(YYINITIAL);
                                  tokenList.add(new Character('\\'));}
  \\[0-3]?{OctDigit}?{OctDigit}\' { yybegin(YYINITIAL);
                                   int val = Integer.parseInt(
                                           yytext().substring(1,yylength()-1),8);
                                   tokenList.add(new Character((char)val)); }

  /* error cases */
  \\.                           { throw new RuntimeException(
                                  "Illegal escape sequence \""+yytext()+"\""); }
  {LineTerminator}              { throw new RuntimeException(
                                  "Unterminated character literal at end of line"); }
}

/* error fallback */
.|\n                            { throw new RuntimeException(
                                   "Illegal character \""+yytext()+
                                   "\" at line "+yyline+", column "+yycolumn); }
<<EOF>>                         { return tokenList;}
```